

# Clase 5 — 25.11.25

#IntelliJIDEA

#JAVA


#XML

#DOM

#SAX

 **Profesor:** Álvaro García Gutiérrez

 **Acceso a Datos**

 **Clase 5 — 25/11/2025**

 **Tema:** Archivos XML | APIS DOM y SAX

## 1 Introducción a XML

XML (*eXtensible Markup Language*) es un **lenguaje de marcas** diseñado para **almacenar y transportar datos** de forma estructurada y legible tanto para humanos como para máquinas. Es un **metalenguaje**, lo que significa que **no define etiquetas fijas**, sino que permite crear las propias según la información que se quiera representar.

Un archivo XML es siempre **texto plano**, lo que lo hace portable entre sistemas y lenguajes.

### ◆ Declaración XML

```
<?xml version="1.0" encoding="UTF-8"?>
```

- **version:** indica la versión del estándar XML.
- **encoding:** define el conjunto de caracteres (habitualmente UTF-8).
- Aunque es **opcional**, si se indica `encoding` debe indicarse también `version`.

📌 Esta línea **debe ser la primera del archivo** si se utiliza.

## 2 Estructura en árbol de un XML

Un XML sigue estrictamente una **estructura jerárquica en forma de árbol**:

- 🌳 **Raíz** → único elemento principal (no puede repetirse).
- 🌲 **Tronco** → elementos intermedios.
- 🌿 **Ramas / hojas** → elementos finales con datos.

Ejemplo conceptual:

```
<raiz>
  <tronco>
    <rama1></rama1>
    <rama2></rama2>
  </tronco>
</raiz>
```

### ◆ Importancia de cerrar correctamente las etiquetas

- Cada etiqueta abierta **debe cerrarse**.
- El orden de cierre **debe ser inverso al de apertura**.
- Un error aquí implica **XML no válido** → ningún parser funcionará.

👉 El profesor insiste en esto porque **DOM y SAX dependen totalmente de esta estructura** para funcionar correctamente.

---

### 3 XML como formato de intercambio de datos

XML no es solo un “formato con etiquetas”, sino un **estándar de facto para representar información estructurada** de forma independiente del lenguaje, del sistema operativo y de la plataforma. Su principal valor está en que **describe datos y su estructura**, no su presentación, lo que lo hace ideal como formato intermedio entre sistemas heterogéneos.

A diferencia de formatos binarios o propietarios, un XML es **autodescriptivo**: cualquier sistema que lo reciba puede interpretar qué significa cada dato simplemente leyendo las etiquetas. Esto explica por qué ha sido tan utilizado históricamente en entornos empresariales, integraciones y sistemas distribuidos.

#### ♦ Comunicación entre aplicaciones

XML se utiliza ampliamente como **lenguaje puente** entre aplicaciones que no comparten tecnología interna. Dos programas pueden estar escritos en lenguajes distintos (Java, C#, Python, PHP) y aun así comunicarse sin problemas si ambos entienden XML.

En este contexto, XML actúa como un **contrato de datos**:

- Define qué información se envía.
- Define cómo se estructura.
- Define qué campos son obligatorios u opcionales.

Por ejemplo, una aplicación puede generar un XML con información de alumnos y otra aplicación, completamente distinta, puede consumirlo sin necesidad de conocer cómo está implementada internamente la primera.

Esta idea es clave en:

- Servicios web clásicos (SOAP).
- Integraciones entre ERPs.
- Intercambio de información entre sistemas legacy y modernos.

---

#### ♦ Persistencia de datos

XML también se utiliza como mecanismo de **almacenamiento estructurado** cuando no se requiere una base de datos relacional completa.

En lugar de guardar datos en tablas, se almacenan en ficheros XML que:

- Mantienen jerarquía.
- Son fácilmente versionables.
- Pueden editarse manualmente si es necesario.
- Se integran bien con control de versiones (Git).

Este enfoque es habitual para:

- Datos de tamaño pequeño o medio.
- Datos jerárquicos naturales.
- Información que debe conservar estructura y significado.

El propio PDF hace énfasis en que XML es **texto plano**, lo que permite:

- Abrirlo con cualquier editor.
- Depurarlo fácilmente.
- Transportarlo sin dependencias externas.

---

## ◆ Configuración de sistemas

Uno de los usos más importantes de XML es la **configuración de aplicaciones**. En este escenario, el XML no transporta datos de negocio, sino **parámetros de funcionamiento**.

Ejemplos habituales:

- Configuración de servidores.
- Definición de rutas, permisos o roles.
- Declaración de módulos activos.
- Inicialización de componentes.

La ventaja frente a otros formatos es que XML permite:

- Estructurar configuraciones complejas.
- Validar la configuración mediante esquemas (XSD).
- Separar claramente código y configuración.

Este uso encaja perfectamente con la filosofía de que **el programa no debe cambiar cuando cambian los datos**.

---

## ◆ Intercambio de información entre lenguajes y plataformas

XML es completamente **agnóstico** respecto al lenguaje de programación. No pertenece a Java, ni a Microsoft, ni a ningún proveedor concreto. Esto lo convierte en una solución ideal para:

- Sistemas distribuidos.
- Arquitecturas cliente-servidor.
- Entornos multiplataforma.
- Integraciones a largo plazo.

Un XML generado hoy puede seguir siendo válido dentro de años, incluso si la aplicación que lo consume ha cambiado de lenguaje o tecnología. Esta estabilidad es uno de los motivos por los que XML ha sido tan adoptado en entornos institucionales y empresariales.

---

## ◆ Ejemplo realista de intercambio de datos

```
<coches>
  <coche>
    <marca>Seat</marca>
    <modelo>Ibiza</modelo>
    <color>rojo</color>
    <matriculacion>2019</matriculacion>
  </coche>
</coches>
```

En este ejemplo se observa claramente:

- Un **elemento raíz único** ( <coches> ).
- Un conjunto de elementos repetibles ( <coche> ).
- Datos encapsulados mediante etiquetas semánticas.

No es necesario ningún comentario externo para entender qué representa cada valor. El propio XML **se explica a sí mismo**, lo cual es una de sus mayores virtudes como formato de intercambio.

---

## 4 Parsers XML en Java

Para poder trabajar con un XML desde Java es imprescindible utilizar un **parser**. Un parser es el componente encargado de **interpretar el texto XML** y transformarlo en una estructura que el programa pueda manejar.

Un parser no se limita a “leer líneas”, sino que realiza varias tareas críticas:

- Analiza la sintaxis del documento.
- Verifica que las etiquetas estén bien formadas.
- Comprueba que la jerarquía sea correcta.
- Expone los datos de forma accesible al código.

Si el XML no cumple las normas básicas (etiquetas mal cerradas, jerarquía incorrecta, errores de estructura), el parser **lanza una excepción** y el documento se considera inválido.

---

### ♦ Qué hace realmente un parser

Desde el punto de vista interno, un parser:

- Lee el fichero XML como un flujo de caracteres.
- Interpreta cada etiqueta, atributo y valor.
- Aplica las reglas del estándar XML.
- Construye un modelo de acceso a los datos (árbol, eventos u objetos).

El parser es el encargado de **verificar la estructura**, no solo de extraer datos. Por eso, antes de trabajar con la información, siempre hay una fase de validación implícita.

---

### ♦ APIs principales para XML en Java

Java ofrece varias APIs para trabajar con XML, cada una con un enfoque distinto:

- **DOM**  
Carga el documento completo en memoria y lo representa como un árbol de objetos. Permite navegación libre, lectura y modificación del XML.
- **SAX**  
Procesa el XML secuencialmente mediante eventos. No construye un árbol ni guarda todo en memoria. Es más eficiente, pero más complejo de programar.
- **JAXB**  
Permite convertir directamente un XML en objetos Java y viceversa. Está orientado a modelos de datos estables y simplifica mucho el código, a costa de exigir un XML bien definido.

🔴 Aunque en esta clase todavía no se trabaja JAXB, es importante entender que **cada parser responde a una necesidad distinta**, y no existe uno “mejor” en términos absolutos.

---

## ♦ Idea clave para la asignatura

El objetivo del profesor no es que memorices APIs, sino que comprendas **cómo piensa cada parser**:

- DOM → “Cargo todo, lo manipulo y luego decido qué hago”.
- SAX → “Leo una vez y reacciono a lo que encuentro”.
- JAXB → “Quiero trabajar con objetos, no con etiquetas”.

Esta comprensión será fundamental cuando se vean ejemplos prácticos y se compare el mismo XML tratado con distintos enfoques.

---

## 5 API DOM (Document Object Model)

### ♦ Concepto general

La API DOM (*Document Object Model*) es un parser XML **orientado a árbol**, lo que significa que, cuando se parsea un archivo XML, **todo su contenido se carga completamente en memoria** y se transforma en una estructura jerárquica de objetos Java. Esta estructura refleja exactamente la forma del XML original: una raíz única, nodos padre, nodos hijos y hojas con datos.

Internamente, DOM convierte cada etiqueta, atributo y fragmento de texto en objetos como `Document`, `Element`, `Node`, `Attr` o `NodeList`. A partir de ese momento, el XML deja de ser “texto” y pasa a ser **un conjunto de objetos manipulables desde Java**.

Esto permite trabajar con el documento de forma muy intuitiva: moverse por el árbol, acceder a un padre desde un hijo, modificar valores, añadir nodos nuevos o eliminar existentes. En la práctica, es como tener el XML **desplegado entero en memoria**, rama por rama, con acceso total a su estructura.

La consecuencia directa de este enfoque es doble:

- Por un lado, DOM es **muy cómodo y expresivo** para el programador.
  - Por otro, implica un **coste elevado en memoria**, especialmente si el XML es grande.
- 

### 5.1 Características principales de DOM

Una de las ideas fundamentales que hay que interiorizar es que DOM no es solo un lector de XML, sino una **representación viva del documento**. Una vez cargado, el documento se puede recorrer tantas veces como se quiera, sin perder información ni estado.

DOM permite una **navegación bidireccional real**. Desde cualquier nodo se puede:

- Subir al nodo padre.
- Bajar a los hijos.
- Moverse entre nodos hermanos.

Esto es especialmente útil cuando se trabaja con estructuras complejas o cuando se necesita acceder a datos que no siguen un orden lineal.

Otra característica clave es que DOM permite **lectura y escritura**. No solo se pueden consultar datos, sino también:

- Modificar textos.
- Cambiar atributos.
- Insertar nuevos elementos.

- Eliminar nodos existentes.

Finalmente, DOM proporciona mecanismos para **guardar los cambios**, transformando de nuevo el árbol en un archivo XML persistente.

Como contrapartida, el parser DOM:

- Consume más memoria, ya que todo el documento está cargado simultáneamente.
- Es más lento que SAX en archivos grandes, porque necesita construir el árbol completo antes de empezar a trabajar.

Por este motivo, DOM es ideal cuando se necesita **control total sobre el documento**, pero no es la mejor opción para XMLs masivos.

---

## **5.2 Secuencia correcta de trabajo en DOM (MUY IMPORTANTE)**

Aunque una vez cargado el XML el árbol DOM permite **navegación libre y bidireccional**, el **proceso de creación, modificación y guardado sigue una secuencia lógica obligatoria**. Este orden no es casual: responde a cómo funciona internamente el parser DOM y es una de las principales fuentes de errores cuando no se entiende bien.

---

### **♦ 1. Creación de la infraestructura del parser**

El primer paso siempre es preparar el entorno que permitirá parsear el XML:

- Se crea una instancia de `DocumentBuilderFactory`.
- A partir de la fábrica se obtiene un `DocumentBuilder`.

En este punto:

- ❌ No hay XML cargado.
- ❌ No existe ningún árbol.
- ✔ Solo se ha preparado el **mecanismo** que permitirá construirlo.

Si esta fase falla, el proceso no puede continuar.

---

### **♦ 2. Parseo del archivo XML → creación del `Document`**

El siguiente paso es parsear el archivo XML:

- El fichero de texto se analiza carácter a carácter.
- Se valida su estructura (etiquetas bien formadas, jerarquía correcta).
- Si todo es correcto, se construye el objeto `Document`.

Aquí ocurre lo más importante:

- 📄 El XML deja de ser texto.
- 🌳 Se convierte en un **árbol DOM completo en memoria**.

Si el XML está mal formado:

- Se lanza una excepción.
  - No se obtiene ningún `Document`.
  - No hay árbol sobre el que trabajar.
-

### ◆ 3. Lectura y recorrido del árbol existente

Una vez existe el `Document`, ya se puede trabajar con él:

- Buscar nodos.
- Recorrer listas ( `NodeList` ).
- Acceder a elementos, atributos y texto.
- Navegar entre padres, hijos y hermanos.

En esta fase:

- El árbol refleja **exactamente** el XML original.
  - No se ha modificado nada todavía.
  - Todas las operaciones son de **lectura**.
- 

### ◆ 4. Creación de nuevos elementos (solo en memoria)

Cuando se crean nuevos elementos usando métodos como `createElement()`:

- Se generan nodos nuevos.
- Se pueden configurar atributos y contenido.
- Se pueden crear hijos y relacionarlos entre sí.

Pero hay una idea clave:

- ⚠ Estos elementos **no pertenecen aún al árbol DOM**.
- ⚠ Existen únicamente como objetos sueltos en memoria.

Crear un elemento **no implica** que ya forme parte del XML.

---

### ◆ 5. Inserción explícita en el árbol DOM

Para que un elemento pase a formar parte real del documento:

- Debe insertarse explícitamente mediante métodos como `appendChild()`.

Este paso:

- Establece la relación padre-hijo.
- Integra el nuevo nodo dentro del árbol.
- Modifica el DOM en memoria.

Hasta que no se ejecuta esta inserción:

- El XML sigue siendo exactamente el mismo.
  - El nuevo elemento es invisible desde el árbol principal.
- 

### ◆ 6. Guardado del documento (persistencia)

Aunque el árbol DOM ya esté modificado, los cambios **solo existen en memoria**.

Para que sean permanentes:

- El árbol debe transformarse de nuevo a texto XML.
- El resultado debe escribirse en un archivo mediante un `Transformer`.

Si este paso no se realiza:

- Al finalizar el programa, todo se pierde.
- El archivo original no se modifica.

---

## Idea clave que resume todo el proceso

### Crear ≠ Insertar ≠ Guardar

- Crear → genera nodos en memoria.
- Insertar → modifica el árbol DOM.
- Guardar → persiste los cambios en un archivo.

Confundir estas fases es uno de los errores más comunes al trabajar con DOM, y por eso el profesor insiste tanto en esta secuencia.

---

### ### 5.3 Creación del árbol DOM

El proceso de creación del árbol DOM comienza con la lectura del archivo XML y su conversión a un objeto `Document`. Este objeto representa el documento completo, incluyendo la raíz, todos los elementos y su jerarquía interna.

```
```java
DocumentBuilderFactory f = DocumentBuilderFactory.newInstance();
DocumentBuilder b = f.newDocumentBuilder();
Document doc = b.parse(new File("alumnos.xml"));
```

En este punto:

- El XML ya está completamente cargado en memoria.
- El árbol DOM ya existe.
- `doc` es el punto de entrada a todo el contenido del documento.

A partir de aquí, cualquier operación de lectura o modificación se realiza **sobre el árbol**, no sobre el archivo directamente.

---

## 5.4 Lectura de datos con DOM

La lectura de datos con DOM se realiza siempre **sobre el árbol que ya está cargado en memoria**, nunca directamente sobre el archivo XML. Esto implica que todas las operaciones de lectura trabajan contra una **estructura estable**, que no cambia a menos que el programa la modifique explícitamente.

---

### ◆ 1. Búsqueda de nodos dentro del árbol

El punto de partida habitual para leer datos es la búsqueda de elementos por su nombre de etiqueta. Para ello se utiliza el método `getElementsByTagName()`.

Este método:

- Recorre todo el árbol DOM.
- Localiza todos los nodos que coinciden con el nombre indicado.
- Devuelve el resultado en forma de `NodeList`.

Es importante entender que la búsqueda:

- No depende de la profundidad del nodo.
- No se limita a un nivel concreto.
- Ignora la posición exacta dentro del XML.

Esto hace que DOM sea muy flexible, pero también exige que el programador **sepa exactamente qué etiquetas está buscando**.

---

## ◆ 2. Uso de `NodeList` como colección de resultados

El objeto `NodeList` actúa como una colección ordenada de nodos:

- Cada posición contiene un nodo encontrado.
- Se accede mediante índice ( `item(i)` ).
- Se puede recorrer con bucles sin perder el orden del XML original.

Cada elemento del `NodeList` es de tipo `Node`, lo que obliga a comprobar o asumir su tipo antes de trabajar con él. En la mayoría de casos prácticos, estos nodos se convierten a `Element` para poder acceder a información más específica.

---

## ◆ 3. Conversión de `Node` a `Element`

Para trabajar cómodamente con los datos del XML, es habitual convertir cada nodo a `Element`.

Esta conversión permite:

- Acceder a los atributos del nodo.
- Buscar elementos hijos.
- Extraer el texto interno de las etiquetas.

Esta fase es fundamental, ya que la interfaz `Node` es muy genérica, mientras que `Element` proporciona métodos específicos para trabajar con etiquetas XML.

---

## ◆ 4. Lectura de atributos y contenido textual

Una vez se trabaja con un `Element`, se pueden extraer distintos tipos de información:

- Los **atributos** se obtienen mediante `getAttribute()`.
- El **contenido textual** de una etiqueta se obtiene con `getTextContent()`.

`getTextContent()` devuelve todo el texto contenido dentro de la etiqueta, incluyendo el de sus nodos hijos si los hubiera. Por eso es importante utilizarlo con conocimiento de la estructura del XML que se está leyendo.

Este enfoque permite acceder tanto a:

- Los **datos** (valores).
  - La **estructura** (relación entre etiquetas).
-

## ◆ 5. Reutilización del árbol DOM

Una de las grandes ventajas de DOM es que el árbol permanece completo en memoria durante toda la ejecución del programa.

Esto implica que:

- Se puede recorrer el documento varias veces.
- Se pueden realizar múltiples búsquedas independientes.
- El orden de lectura no afecta al resultado.
- No es necesario volver a parsear el XML para cada operación.

Mientras no se modifique explícitamente el árbol, el contenido se mantiene intacto, lo que hace de DOM una opción muy potente cuando se necesitan **lecturas complejas o repetidas** sobre el mismo documento.

### 5.5 Creación e inserción de elementos (idea clave de la clase)

Uno de los conceptos más importantes de esta clase es entender que **DOM trabaja siempre en memoria** y que las modificaciones siguen una lógica muy concreta.

Cuando se crea un nuevo elemento con `createElement()`, se está generando un nodo aislado que todavía no pertenece al documento. Se le pueden añadir atributos, texto y nodos hijos sin ningún problema, pero sigue siendo un elemento “flotante”.

```
Element nuevo = doc.createElement("alumno");
nuevo.setAttribute("id", "A03");
```

Del mismo modo, los elementos hijos se crean y configuran de forma independiente:

```
Element nombre = doc.createElement("nombre");
nombre.setTextContent("Carmen");
```

Solo cuando se utiliza `appendChild()` se establece una relación real dentro del árbol DOM. Primero se enlazan los hijos con el padre, y después el nuevo nodo se inserta en el árbol principal, normalmente colgándolo del elemento raíz.

```
nuevo.appendChild(nombre);
doc.getDocumentElement().appendChild(nuevo);
```

🔴 Hasta este último paso, el XML original no ha cambiado en absoluto.

🔴 El orden de estas operaciones es fundamental para que el documento tenga sentido estructural.

### 5.6 Guardado del XML modificado

Una vez que el árbol DOM ha sido modificado, los cambios solo existen en memoria. Para que estos cambios se reflejen en un archivo XML es necesario realizar una **transformación inversa**, convirtiendo el árbol en texto.

Para ello se utiliza la API `Transformer`, que toma como fuente el objeto `Document` y como destino un archivo o un flujo de salida. Este proceso serializa el árbol DOM y genera un nuevo XML, ya con las modificaciones aplicadas.

Este paso final cierra el ciclo completo de DOM:

- Texto XML → árbol en memoria.
- Manipulación del árbol.
- Árbol en memoria → texto XML persistente.

## 6 Ejemplo práctico DOM

En este apartado se documenta **exactamente** el ejemplo visto en clase:

`alumnos.xml + DOM.java`

El objetivo del ejercicio es **entender cómo funciona DOM internamente**, insistiendo en:

- la **estructura en árbol**,
- la **secuencia correcta de acciones**,
- y la diferencia entre **crear**, **insertar** y **guardar**.

### 6.1 Archivo XML de entrada — `alumnos.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<grupo curso="2DAM" tutor="Jose Luis">

  <alumno id="A01">
    <nombre>Fran</nombre>
    <edad>20</edad>
    <modulos>
      <modulo>Acceso a Datos</modulo>
      <modulo>Desarrollo de Interfaces</modulo>
    </modulos>
  </alumno>

  <alumno id="A02">
    <nombre>Sonia</nombre>
    <edad>21</edad>
    <modulos>
      <modulo>PSP</modulo>
      <modulo>Acceso a Datos</modulo>
    </modulos>
  </alumno>

  <alumno id="A03">
    <nombre>Javier</nombre>
    <edad>22</edad>
  </alumno>

</grupo>
```

#### Observaciones importantes sobre el XML

- `<grupo>` es el **elemento raíz** (tronco).
- `<alumno>` son **ramas repetibles**.

- `id` es un **atributo** del elemento `<alumno>`.
- `<nombre>`, `<edad>` y `<modulos>` son **elementos hijos**.
- La jerarquía está **perfectamente cerrada**, requisito imprescindible para DOM.

## 6.2 Archivo Java — DOM.java (código comentado)

```
package DOM;

import org.w3c.dom.*;
// Modelo DOM: Document (árbol completo), Element (etiquetas), Node, NodeList...

import javax.xml.parsers.DocumentBuilder;
// Parser DOM: construye el Document a partir del XML

import javax.xml.parsers.DocumentBuilderFactory;
// Fábrica que crea y configura parsers DOM

import javax.xml.transform.*;
// API para transformar el árbol DOM en XML (guardar cambios)

import javax.xml.transform.dom.DOMSource;
// Fuente de la transformación: el Document en memoria

import javax.xml.transform.stream.StreamResult;
// Destino de la transformación: archivo XML

import java.io.File;
// Manejo de archivos

public class DOM {

    public static void main(String[] args) throws Exception {

        // =====
        // 1 Crear la infraestructura del parser DOM
        // =====

        DocumentBuilderFactory f = DocumentBuilderFactory.newInstance();
        // Se crea la fábrica de parsers DOM.
        // Aquí todavía NO hay ningún XML cargado.

        DocumentBuilder b = f.newDocumentBuilder();
        // A partir de la fábrica se obtiene el parser DOM.
        // El parser está listo para parsear XML.

        // =====
        // 2 Parsear el XML → creación del árbol DOM
        // =====

        Document doc = b.parse(new File("alumnos.xml"));
        // En este punto:
        // - El XML deja de ser texto
```

```

// - Se valida su estructura
// - Se construye el árbol DOM completo en memoria
// 'doc' representa TODO el XML

// =====
// 3 Lectura del XML (recorrido del árbol)
// =====

NodeList alumnos = doc.getElementsByTagName("alumno");
// Busca TODOS los nodos <alumno> del árbol
// No importa su profundidad ni posición

for (int i = 0; i < alumnos.getLength(); i++) {

    Element e = (Element) alumnos.item(i);
    // Cada item del NodeList es un Node
    // Se convierte a Element para acceder a atributos y etiquetas

    String id = e.getAttribute("id");
    // Lee el atributo id="A01", "A02", etc.

    String nombre = e.getElementsByTagName("nombre")
        .item(0)
        .getTextContent();
    // Busca el primer <nombre> hijo
    // Extrae el texto interno de la etiqueta

    System.out.println(id + ": " + nombre);
    // Salida por consola: A01: Fran
}

// =====
// 4 Crear un nuevo alumno (SOLO en memoria)
// =====

Element nuevo = doc.createElement("alumno");
// Se crea un nuevo nodo <alumno>
// IMPORTANTE: todavía NO forma parte del árbol DOM

nuevo.setAttribute("id", "A04");
// Se añade el atributo id="A04"

Element nom = doc.createElement("nombre");
nom.setTextContent("Carmen");
// Se crea <nombre>Carmen</nombre>

Element edad = doc.createElement("edad");
edad.setTextContent("23");
// Se crea <edad>23</edad>

// =====
// 5 Inserción en el árbol DOM
// =====

```

```

nuevo.appendChild(nom);
// <nombre> pasa a ser hijo de <alumno>

nuevo.appendChild(edad);
// <edad> pasa a ser hijo de <alumno>

doc.getDocumentElement().appendChild(nuevo);
// El nuevo <alumno> se cuelga del elemento raíz <grupo>
// AHORA sí forma parte real del árbol DOM

// =====
// 6 Guardar el DOM modificado a un archivo
// =====

Transformer t = TransformerFactory
    .newInstance()
    .newTransformer();
// Crea el transformador DOM → XML

t.setOutputProperty(OutputKeys.INDENT, "yes");
// Activa el sangrado para que el XML sea legible

t.transform(
    new DOMSource(doc),
    // Fuente: el árbol DOM en memoria
    new StreamResult(new File("alumnos_out.xml"))
    // Destino: archivo alumnos_out.xml
);
}
}

```

## 6.3 Análisis detallado por bloques

### ◆ DocumentBuilderFactory y DocumentBuilder

- DocumentBuilderFactory

Es el punto de entrada al mundo DOM. Permite configurar cómo se va a crear el parser.

- DocumentBuilder

Es el parser real. Sin él no se puede convertir el XML en un árbol.

📌 Hasta aquí **no existe ningún XML en memoria**.

### ◆ parse(File)

- Es el **punto crítico** del proceso.
- Convierte texto XML → **árbol DOM**.
- Si el XML está mal cerrado o mal formado → excepción.
- Devuelve un Document, que representa **todo el fichero**.

📌 A partir de aquí **ya no se trabaja con el archivo**, sino con el árbol.

### ◆ getElementsByTagName()

- Busca nodos por nombre de etiqueta.
- Recorre **todo el árbol**.
- Devuelve un `NodeList` ordenado según el XML original.

📌 No es una búsqueda “lineal”, es una **consulta al árbol en memoria**.

---

#### ◆ **Node → Element**

- `Node` es genérico (puede ser texto, comentario, elemento...).
- `Element` permite:
  - leer atributos ( `getAttribute` )
  - buscar hijos
  - extraer contenido

Por eso siempre se hace el *casting*.

---

#### ◆ **createElement() vs appendChild()**

Este es **EL concepto clave del ejercicio**:

- `createElement()`  
→ crea nodos **sueltos**, solo en memoria.
- `appendChild()`  
→ **conecta nodos** y los inserta en el árbol DOM.

📌 Si no hay `appendChild()` , el XML **no cambia**.

---

#### ◆ **Transformer**

- Convierte el árbol DOM de vuelta a texto XML.
- Sin este paso:
  - los cambios existen solo en memoria
  - al terminar el programa se pierden

📌 DOM siempre sigue el ciclo:

**XML → árbol → modificación → XML**

---

## 7 API SAX (Simple API for XML)

En contraposición al enfoque de DOM, la API SAX propone una forma **radicalmente distinta** de trabajar con XML. Mientras que DOM transforma el documento completo en una estructura de árbol en memoria, SAX adopta un enfoque **basado en flujo (streaming)**, en el que el XML se procesa **secuencialmente, de principio a fin**, sin conservar una representación completa del documento.

Esto implica un cambio importante de mentalidad:

con SAX no se “consulta” un documento, sino que se **escucha cómo se va leyendo**. El parser avanza por el XML y va notificando al programa cada evento relevante que detecta. El control del recorrido **no lo tiene el programador**, sino el propio parser.

---

## ◆ Concepto general de SAX

SAX (*Simple API for XML*) es un parser que se caracteriza por:

- Estar **orientado a eventos**, no a estructuras.
- Procesar el XML de forma **secuencial**.
- No construir ningún árbol en memoria.
- Tener un **consumo de memoria muy bajo**.
- Ser especialmente eficiente con documentos grandes.

Cuando SAX procesa un XML:

- No guarda el documento completo.
- No permite retroceder ni “volver a un nodo anterior”.
- No ofrece navegación padre/hijo/hermano.
- No permite modificar directamente la estructura del XML.

✚ Por diseño, SAX está pensado para **leer y reaccionar**, no para editar.

Esta filosofía lo convierte en una herramienta muy adecuada para escenarios donde el XML es grande o se recibe como flujo (por ejemplo, desde red o desde un sistema externo).

---

## ◆ Funcionamiento interno de SAX

Internamente, SAX funciona como un **lector de eventos**:

- El parser lee el XML carácter a carácter.
- Identifica patrones sintácticos (etiquetas, texto, atributos).
- Cada vez que detecta algo relevante, **lanza un evento**.
- Ese evento provoca la ejecución automática de un método del **handler**.

El flujo de control es inverso al habitual en programación imperativa:

- En DOM, el programador llama a métodos para recorrer el árbol.
- En SAX, el parser llama a los métodos del programador.

Por eso se dice que SAX sigue un modelo **reactivo**:

| el código no decide cuándo leer, sino cómo reaccionar cuando algo se lee.

---

## ◆ Eventos principales en SAX

Durante el procesamiento de un XML, SAX genera una serie de eventos bien definidos, entre los que destacan:

- Inicio del documento.
- Inicio de una etiqueta.
- Texto contenido entre etiquetas.
- Fin de una etiqueta.
- Fin del documento.

Cada uno de estos eventos está asociado a un método que puede sobrescribirse. Si no se sobrescribe, simplemente no se hace nada cuando ocurre ese evento.

Este modelo permite un control muy fino del procesado, pero obliga al programador a **gestionar manualmente el contexto**.

---

### ♦ Papel del `DefaultHandler`

Para trabajar con SAX en Java es necesario crear una clase auxiliar que **extienda de `DefaultHandler`**. Esta clase actúa como el **punto de entrada de todos los eventos SAX**: el parser no devuelve datos, sino que *llama* a los métodos de este handler a medida que avanza por el XML.

`DefaultHandler` proporciona implementaciones vacías de los métodos más importantes, lo que permite sobrescribir **solo aquellos que interesan** para el caso concreto.

Ejemplo mínimo de un handler SAX:

```
import org.xml.sax.helpers.DefaultHandler;

public class SaxHandler extends DefaultHandler {
    // Aquí se sobrescriben los métodos necesarios
}
```

🔗 Sin esta clase auxiliar, SAX **no tiene dónde notificar** lo que va leyendo y el parser no puede ser utilizado de forma práctica.

---

### ♦ Método `startElement()`

`startElement()` se ejecuta **cada vez que el parser encuentra una etiqueta de apertura** (`<etiqueta>`).

Es el lugar donde se suele:

- Identificar el nombre del elemento actual.
- Leer atributos.
- Activar variables de control (booleanos).
- Preparar el contexto para el texto que vendrá después.

Ejemplo típico:

```
@Override
public void startElement(String uri, String localName, String qName, Attributes
attributes) {

    if (qName.equalsIgnoreCase("nombre")) {
        esNombre = true;
    }

    if (qName.equalsIgnoreCase("alumno")) {
        String id = attributes.getValue("id");
        System.out.println("Alumno ID: " + id);
    }
}
```

🔗 En este método **no se procesa el contenido textual**, solo se detecta *qué* empieza.

---

## ♦ Método `characters()`

`characters()` se ejecuta cuando SAX encuentra **texto entre etiquetas**.

Recibe:

- Un array de caracteres ( `char[] ch` )
- Un índice de inicio ( `start` )
- Una longitud ( `length` )

El texto real debe reconstruirse manualmente a partir de esos datos.

Ejemplo:

```
@Override
public void characters(char[] ch, int start, int length) {

    if (esNombre) {
        String nombre = new String(ch, start, length);
        System.out.println("Nombre: " + nombre);
        esNombre = false;
    }
}
```

🔴 Si no se procesa el texto aquí, **se pierde**, porque SAX no lo almacena.

🔴 Además, `characters()` puede llamarse varias veces para un mismo texto, lo que obliga a ser cuidadoso en XML más complejos.

---

## ♦ Método `endElement()`

`endElement()` se ejecuta cuando el parser encuentra una **etiqueta de cierre** (`</etiqueta>`).

Se utiliza para:

- Indicar que un bloque lógico ha terminado.
- Resetear estados.
- Separar visualmente o lógicamente los datos leídos.

Ejemplo:

```
@Override
public void endElement(String uri, String localName, String qName) {

    if (qName.equalsIgnoreCase("alumno")) {
        System.out.println("Fin de alumno");
        System.out.println("-----");
    }
}
```

🔴 En SAX no se “sube” al padre: simplemente se notifica que algo ha terminado y el parser continúa.

---

## ♦ Uso de variables de control (booleanos)

Como SAX no mantiene contexto estructural, el programador debe **simularlo manualmente**. La técnica más habitual es el uso de variables booleanas.

Declaración típica:

```
boolean esNombre = false;
boolean esEdad = false;
```

Activación en `startElement()`:

```
if (qName.equalsIgnoreCase("edad")) {
    esEdad = true;
}
```

Uso en `characters()`:

```
if (esEdad) {
    System.out.println("Edad: " + new String(ch, start, length));
    esEdad = false;
}
```

Este patrón permite saber **qué texto pertenece a qué etiqueta**, algo que en DOM viene dado automáticamente por el árbol.

---

## ◆ Lectura de atributos en SAX

Los atributos solo están disponibles en `startElement()` a través del objeto `Attributes`.

Ejemplo de lectura segura:

```
String id = attributes.getValue("id");
if (id != null) {
    System.out.println("ID del alumno: " + id);
}
```

- Si el atributo existe → devuelve su valor.
- Si no existe → devuelve `null`.

🔴 Esto obliga a escribir código defensivo, pero evita errores silenciosos.

---

## ◆ Secuencia típica de eventos en SAX (con código mental)

Para un XML como:

```
<alumno id="A01">
  <nombre>Fran</nombre>
</alumno>
```

SAX ejecutará internamente algo equivalente a:

```
startElement("alumno")
startElement("nombre")
```

```
characters("Fran")
endElement("nombre")
endElement("alumno")
```

🔴 Cada evento ocurre **una sola vez**, en orden, sin posibilidad de volver atrás.

---

## ♦ Excepciones comunes en SAX

El PDF destaca que el uso de SAX implica manejar varias excepciones:

- `SAXException` y `SAXParseException`  
Asociadas a errores de sintaxis o estructura del XML.
- `IOException`  
Relacionadas con problemas de entrada/salida durante la lectura.
- `ParserConfigurationException`  
Vinculadas a errores al configurar o construir el parser.

Estas excepciones refuerzan la idea de que SAX es **estricto** y no tolera XML mal formado.

---

## ♦ Cuándo usar SAX

SAX es especialmente adecuado cuando:

- El XML es muy grande.
- Se necesita procesar el documento una sola vez.
- Solo interesa una parte de la información.
- El rendimiento y el consumo de memoria son críticos.
- El XML se recibe como flujo (streaming).

En estos casos, DOM supone una sobrecarga innecesaria.

---

Perfecto. Amplíe **sin cambiar el código**, centrándome en **entender de verdad qué está pasando internamente**, que es exactamente lo que el profesor quiere que interioricéis.  
Lo dejo en **formato Obsidian**, con texto más explicativo y menos esquemático.

---

## 8 Ejemplo práctico SAX — Lectura secuencial con `coches.xml`

Este ejemplo de SAX no pretende que memorices código, sino que **comprendas el flujo real de ejecución**.

El profesor insiste en que SAX es “extraño” porque **no sigue el orden visual del archivo**, sino un **orden de eventos internos** que se repite continuamente.

Aquí se trabaja con el archivo `coches.xml`, leyendo su contenido **sin crear ningún árbol en memoria**.

---

### `SaxMain1.java` — Lanzador del parser SAX

```
package SAX;

import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
```

```

public class SaxMain1 {

    public static void main(String[] args) {

        try {
            // 1 Crear la fábrica SAX (patrón Factory)
            // No se carga ningún XML todavía
            SAXParserFactory factory = SAXParserFactory.newInstance();

            // 2 Obtener un parser SAX concreto
            // La implementación real depende del sistema/JDK
            SAXParser saxParser = factory.newSAXParser();

            // 3 Crear el handler (gestor de eventos)
            // Aquí NO se parsea nada aún
            SaxHelper1 handler = new SaxHelper1();

            // 4 Lanzar el parseo
            // A partir de aquí el control pasa al parser
            // El parser leerá el XML y llamará a los métodos del handler
            saxParser.parse("coches.xml", handler);

        } catch (Exception e) {
            // 5 Cualquier error de E/S o de parseo termina aquí
            e.printStackTrace();
        }
    }
}

```

### Clave conceptual de este archivo

- Este `main` **no recorre el XML**.
- No hay bucles ni acceso a nodos.
- Su única función es:
  - crear el parser
  - crear el handler
  - disparar el parseo

 **Toda la lógica real está en el handler.**

### `SaxHelper1.java` — Handler SAX (núcleo del ejemplo)

```

package SAX;

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class SaxHelper1 extends DefaultHandler {

    // =====
    // Estado interno (simula el contexto del árbol DOM)

```

```
// =====

boolean esMarca = false;
boolean esModelo = false;
boolean esColor = false;
boolean esMatriculacion = false;

// Contador de coches completos
int totalCoches = 0;

// =====
// 1 startElement – apertura de etiqueta
// =====

@Override
public void startElement(String uri,
                        String localName,
                        String elementos,
                        Attributes atributos)
    throws SAXException {

    // Mensaje puramente didáctico para ver el flujo SAX
    System.out.println("Inicio del elemento: " + elementos);

    // Activamos banderas según la etiqueta que se abre
    switch (elementos) {

        case "marca":
            // A partir de ahora esperamos texto de <marca>
            esMarca = true;
            break;

        case "modelo":
            esModelo = true;
            break;

        case "color":
            esColor = true;
            break;

        case "matriculacion":
            esMatriculacion = true;
            break;

        default:
            // Otras etiquetas (coches, coche...) no se procesan aquí
            break;
    }
}

// =====
// 2 characters – texto entre etiquetas
// =====
```

```

@Override
public void characters(char[] ch,
                      int inicio,
                      int length)
    throws SAXException {

    // Convertimos el tramo de caracteres en String
    // y eliminamos espacios/saltos irrelevantes
    String texto = new String(ch, inicio, length).trim();

    // Si solo hay espacios o saltos de línea, no hacemos nada
    if (texto.isEmpty()) return;

    // Según la bandera activa, sabemos a qué etiqueta pertenece el texto
    if (esMarca) {
        System.out.println("Marca: " + texto);
        esMarca = false;          // Apagamos la bandera
        return;                   // Evitamos doble procesamiento
    }

    if (esModelo) {
        System.out.println("Modelo: " + texto);
        esModelo = false;
        return;
    }

    if (esColor) {
        System.out.println("Color: " + texto);
        esColor = false;
        return;
    }

    if (esMatriculacion) {
        System.out.println("Matriculación: " + texto);
        esMatriculacion = false;
    }
}

// =====
// 3 endElement – cierre de etiqueta
// =====

@Override
public void endElement(String uri,
                      String localName,
                      String elementos)
    throws SAXException {

    // Mensaje didáctico para ver cuándo se cierra una etiqueta
    System.out.println("Fin del elemento: " + elementos);

    // Cuando se cierra </coche>, sabemos que hemos leído un coche completo
    if ("coche".equals(elementos)) {
        totalCoches++;
    }
}

```

```
}  
}  
}
```

## 🧩 Cómo encaja este código con la explicación teórica

- `startElement()`  
→ detecta **qué empieza** y activa contexto
- `characters()`  
→ procesa **el texto justo cuando llega**  
→ si no se procesa aquí, se pierde
- `endElement()`  
→ marca **el final lógico** de una estructura

🔗 El flujo real es siempre:

**startElement** → **characters** → **endElement**  
y vuelve a empezar **cada vez que se abre una etiqueta**

## 8.1 Qué ocurre realmente cuando se llama a `parse()`

Cuando ejecutamos:

```
saxParser.parse("coches.xml", handler);
```

ocurren varias cosas importantes que **no se ven en el código**:

- El parser abre el archivo XML como un flujo de datos.
- Empieza a leer carácter a carácter.
- Cada vez que reconoce una estructura XML válida:
  - Llama automáticamente a un método del `handler`.
- El programador **no controla el orden** ni decide cuándo se llama a cada método.

🔗 A partir de este punto, el programa **ya no manda**.

El parser es quien dirige la ejecución.

## 8.2 Por qué `startElement()` se ejecuta tantas veces

Cada vez que SAX encuentra una etiqueta de apertura ( `< . . >` ), ejecuta:

```
startElement(...)
```

Esto incluye:

- El elemento raíz ( `<coches>` )
- Cada `<coche>`
- Cada etiqueta interna ( `<marca>` , `<modelo>` , etc.)

Por eso, el parámetro `elements` (`qName`):

- Puede valer `"coches"`

- Puede valer "coche"
- Puede valer "marca", "modelo", etc.

📌 **No existe distinción entre tronco y ramas** para SAX:

todo son eventos de apertura y cierre.

---

### 8.3 El papel real de las banderas booleanas

En DOM, el árbol mantiene el contexto automáticamente.

En SAX **no existe contexto**, por lo que el programador debe **crearlo manualmente**.

Cuando en `startElement()` hacemos:

```
esMarca = true;
```

lo que estamos diciendo realmente es:

“El próximo texto que aparezca pertenece a la etiqueta `<marca>`”

Ese estado:

- Se mantiene activo hasta que `characters()` lo usa.
- Se apaga manualmente después de procesar el texto.

📌 Si no se usan estas banderas:

- No sabríamos a qué etiqueta pertenece el texto.
- El contenido llegaría “sin contexto”.

---

### 8.4 Por qué `characters()` es el método más delicado

`characters()` no se ejecuta “una vez por etiqueta”.

SAX puede llamar a este método:

- Varias veces para el mismo texto.
- Incluso separando palabras o números.

Por ejemplo, el texto:

```
<marca>Seat</marca>
```

podría generar internamente:

- `characters("Se")`
- `characters("at")`

Por eso el código:

- Hace `trim()`
- Ignora textos vacíos
- Apaga la bandera tras procesar el contenido

📌 Este comportamiento explica por qué SAX es más complejo que DOM, pero también más eficiente.

---

## 8.5 Por qué `endElement()` es tan importante

Cuando se ejecuta:

```
endElement(...)
```

el parser está diciendo:

“No vendrá más texto para esta etiqueta”

Este momento es clave para:

- Cerrar bloques lógicos.
- Contar elementos completos (como los coches).
- Resetear estados si fuese necesario.

En el ejemplo:

```
if ("coche".equals(elementos)) {  
    totalCoches++;  
}
```

El contador se incrementa **solo cuando el coche está completamente leído**, no antes.

---

## 8.6 El “bucle invisible” de SAX (idea central de la clase)

Aunque el archivo XML se vea lineal, SAX funciona como un **bucle de eventos**:

Para cada etiqueta:

- 1 Se entra en `startElement()`
- 2 Se pasa por `characters()` si hay texto
- 3 Se sale por `endElement()`

Y este ciclo se repite **una y otra vez**, para cada etiqueta que se abre.

🔴 El flujo **no sigue el código de arriba a abajo**.

El flujo **salta constantemente** entre métodos.

Esto es lo que el profesor llama:

“La forma rara de funcionar del SAX”

---

## 8.7 Diferencia mental clave respecto a DOM

- DOM:
  - Primero construye todo.
  - Luego navegas tranquilamente.
  - El orden lo decides tú.
- SAX:
  - Nunca construye nada.
  - No navegas.
  - Reaccionas a lo que llega.

- El orden lo impone el parser.

✚ SAX no es más difícil por capricho,  
es más **estricto** y **determinista**.

---